

Automated Fault Tree Analysis

Johann Deneux

November 19, 2001

Abstract

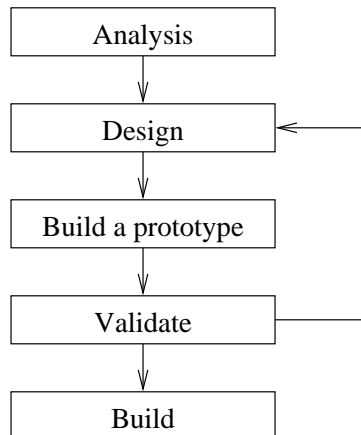
Fault Tree Analysis (FTA) is an important part of the design of systems. The aim of FTA is to find causes of system errors already during the design stage. We introduce a new method using SAT-solvers, which can be used to analyze monotone systems. For other non-monotone systems, we show an existing method using BDDs. Finally, we present a few extensions to traditional FTA.

1 Introduction

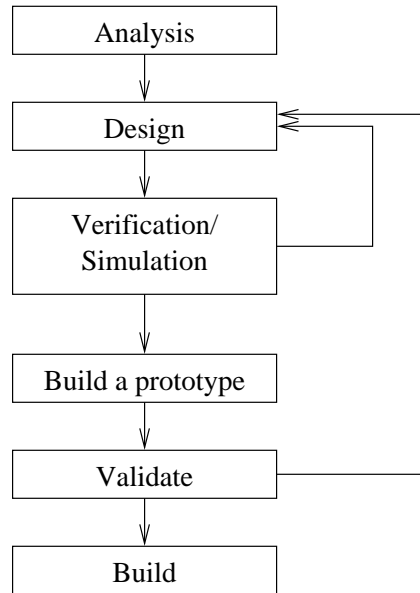
We present what FTA consists of, why it is needed, and how to perform such an analysis.

This type of analysis was introduced in the forties, and benefited from the recent advances in the field of formal methods.

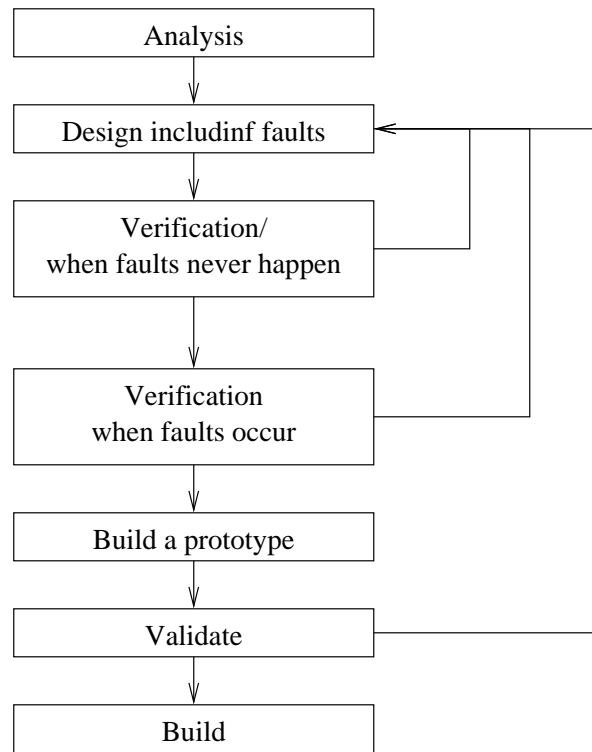
Below is a figure showing the “traditional” development cycle of a product. The analysis stage consists of an informal definition of what the product is supposed to do. Then comes the design phase, followed by the construction of a prototype to verify and find caveats in the design. Once this is done, the design is corrected, and a new prototype is built.



Building and testing prototypes can be both time and money consuming. Therefore, it is usually a good idea to perform an amount of testing and verification in early design stages, before any prototype is actually made. This is the step where formal methods are involved.



The process described above analyzes correctness, under the assumption the all of its components are *perfect*, i.e. never fail. However this assumption is not very realistic. It is often vital to prevent at least some of the most common failures. If system designers know what failures a system is likely to encounter, they can try to modify their design to prevent these failures. If such a correction is not possible or feasible, they can at least try to reduce the probability of occurrence probable by using redundancy. It is also possible to create procedures to fix faulty systems as quickly and cheaply as possible.



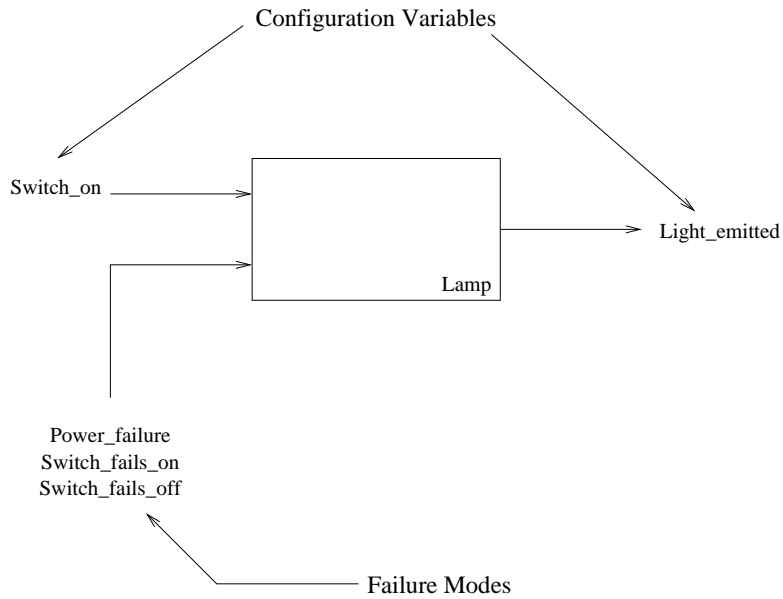
In the above process, a model is built during the design phase. It includes some of the imperfections of the components. For example, a battery could fail to provide electricity. The first verification step ensures that the system works correctly when no failure occurs. Then, we analyze the system when failures are allowed to happen.

2 Fault Tree analysis

In order to be able to simulate and perform verifications on a system, it must be described in a precise way. The information needed is the following:

- Inputs of the system
- Outputs
- How inputs and outputs are related

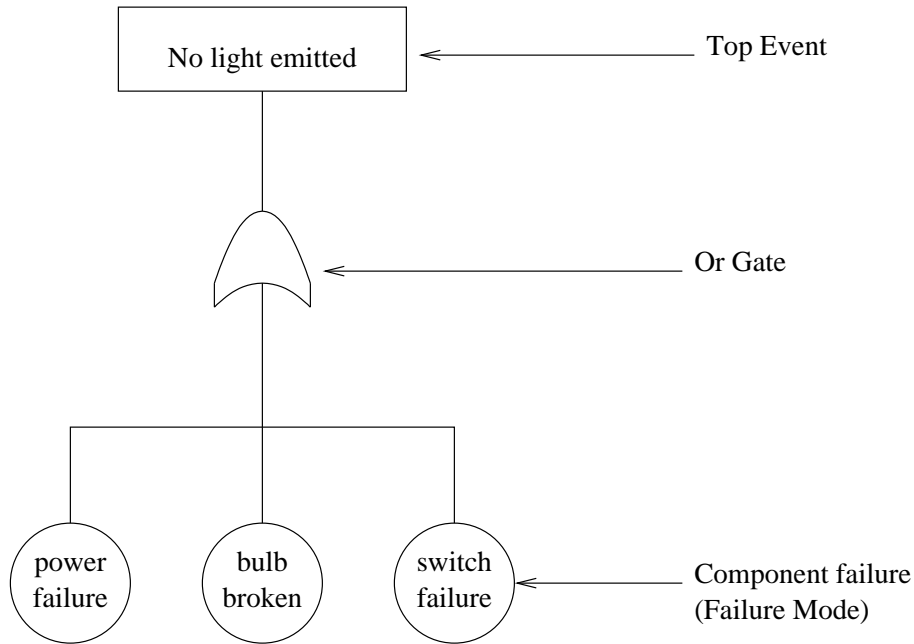
Each input and output is called a *variable* of the system. We split the variables in two groups: *configuration* variables, and *failure modes*.



Configuration variables are used to model interactions with users (or other systems). A *configuration* of a system is an assignment of values to all the configuration variables. Failure modes play a special role: they trigger failures of components and cannot be controlled by the system or the user. In fact, they are enabled randomly.

While Event Tree Analysis deals with finding the consequences of a component-level failure, FTA consists of finding causes of system-level failures.

Here we will talk only about the later analysis, which involves Fault Trees.



Fault Tree

A Fault Tree is a structure exposing the relationships between failure modes and system-level failures. It is a tree representing a boolean expression. Its leaves are *failures modes*, its root represents the *Top Event*.

3 Monotone systems

First, we will describe informally monotone systems. Then, we will define monotone boolean functions. Finally, we will present an algorithm based on a SAT-Solver to perform FTA automatically.

3.1 Informal description

An example of a monotone system is given in the previous section. It illustrates the terms failure modes, configuration variables. One of the fault trees describing the system failures of this lamp illustrates the definition of a fault tree.

In this example, if the bulb brakes, no light can be emitted. Similarly, a power failure prevents the emission of light. If both a power and a bulb failure occur, no light can still be emitted. It is sufficient to have one of these component fail to induce a system failure.

3.2 Definition

As Fault Trees represent boolean expressions, vocabulary dealing with boolean functions is introduced below.

Monotone boolean function

Let A and B be two words of n bits. We have $A \subseteq B$ iff $\forall i : A_i \Rightarrow B_i$. A boolean function f is said to be monotone iff $A \subseteq B \Rightarrow f(A) \subseteq f(B)$

For example, if a boolean expression contains “AND” and/or “OR”, but no “NOT” nodes, it is *monotone*. Knowing that a function f is monotone is helpful. Indeed, if we know for example that f is true when the variable x is true and y is false, we can deduce that f will also be true when y is true. This leads us to the definition of a cut set.

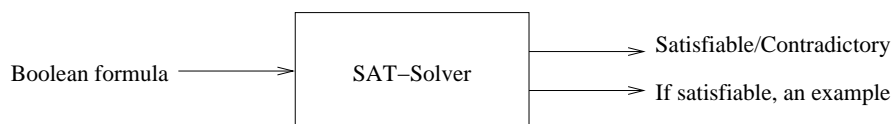
Cut set of a monotone function

A cut set is a set of variables, which, when set to true, make the function true. Notice that if $\{A, B, C\}$ is a cut set, then so are $\{A, B, C, D\}$, $\{A, B, C, E\}$, $\{A, B, C, D, E\}$... A cut set is *minimal* when none of its subsets is also a cut set.

Finding the combination of failure modes inducing a system-level failure is an instance of the problem of finding the minimal cut sets of a boolean function.

3.3 Solution using a SAT solver

SAT solvers are tools used to check the satisfiability of boolean expressions. Given a boolean formula, such tools reply either “Yes, this formula is satisfiable, and an example is ...” or “No, this formula is contradictory”.



An example of how to use a SAT solver in FTA is given in [1]. Here is a generalization of this algorithm, which can be used to find all minimal cut sets (MCS) of any monotone boolean function.

The idea is to extract minimal cut sets from satisfying assignments, by increasing size. Each iteration of the outer loop computes an example satisfying the boolean function f . Each such example contains a cut set of size k .

```
Function ComputeMCS( Boolean Function f ): Set of Minimal Cut Sets
Variables
  Boolean expression previouslyExplored := FALSE
  Set of Minimal Cut Sets result := empty set
  Boolean expression formula
```

```

Integer n := number of variables of f
Integer k;
Boolean satisfiable
Minimal Cut Set mcs
Begin
  For k := 0 to n
    Repeat
      formula := Eqk(k, x1, x2, ..., xn) AND NOT f(x1, x2, ..., xn)
                AND NOT previouslyExplored
      (satisfiable, example) := CallSATSolver(formula)

      If (satisfiable) then
{The set of variables set to TRUE in example is a MCS}
        mcs = ExtractPositiveVariables(example)

        previouslyExplored := previouslyExplored OR VAR(mcs)
      EndIf
    Until not satisfiable
  next k

  Return result
EndFunction

{ Build the conjunction of a set of variables }
Function VAR(Set of variables mcs): Boolean expression
Variables
  Boolean expression ReturnedExpression := TRUE
Begin

  Forall variable X of mcs
    ReturnedExpression := ReturnedExpression AND X
  Done

  Return ReturnedExpression

EndFunction

```

We can use this algorithm for FTA by replacing f by a requirement, i.e. a boolean formula expressing a property a system must respect. For example, it could be “The bulb lights on when the lamp is switched on”. A requirement usually binds together several variables: failure variables, inputs and outputs. Here, we fix the values of inputs and outputs. The analysis is performed for a given configuration of the system.

Does this algorithm really solve our problem ? We need to verify that it really does three things:

- When it terminates, do we get all the minimal cut sets ?
- When it terminates, do we get only the minimal cut sets ?
- Does it always terminate ?

3.3.1 All minimal cut sets are returned

We will use a proof by contradiction. Suppose that the algorithm misses one minimal cut-set C of size l . As the algorithm terminated, *formula* became contradictory for $k = l$. That means no assignment of values to the variables could make f true. However, C verifies $Eqk(k, x1, x2, \dots, xn)$ (because $k = l$). Moreover, it satisfies $NOT f(x1, x2, \dots, xn)$ (otherwise it would not be a cut set). Finally, m does not contradict $NOT PreviouslyExplored$, as it was not found. But that means *formula* can be satisfied, what is a contradiction.

3.3.2 Only minimal cut sets are retrieved

Here again, we will use a proof by contradiction. Assume the SAT-Solver returned a non-minimal cut set C . Then there must be a minimal cut set M such that $M \subseteq C$. As all minimal cut sets are computed, M is in *PreviouslyExplored*. Therefore, C contradicts *PreviouslyExplored*, and could not be returned by the SAT-Solver.

3.3.3 The algorithm terminates

The *for* loop terminates iff the *Repeat* does, for every k . To prove that, we can try to find an expression that must remain positive, and that is decreased by every iteration of the *Repeat* loop. Such an expression is the number of minimal cut sets of size k not yet found. Indeed, there are a finite number of those, and one of them is removed at each iteration.

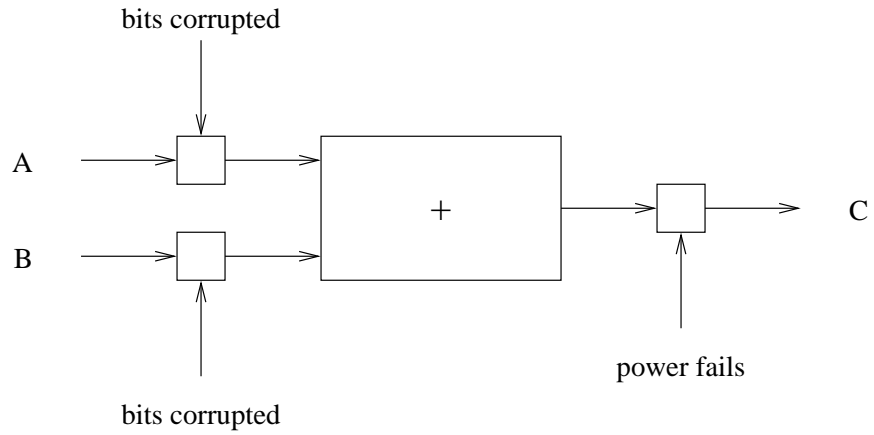
This algorithm has an interesting feature: it computes the cut sets in an iterative manner. As designers are often interested in the smaller cut sets only, it could be possible to stop the computation once the designer is satisfied with the cut sets already computed.

4 Non-monotone systems

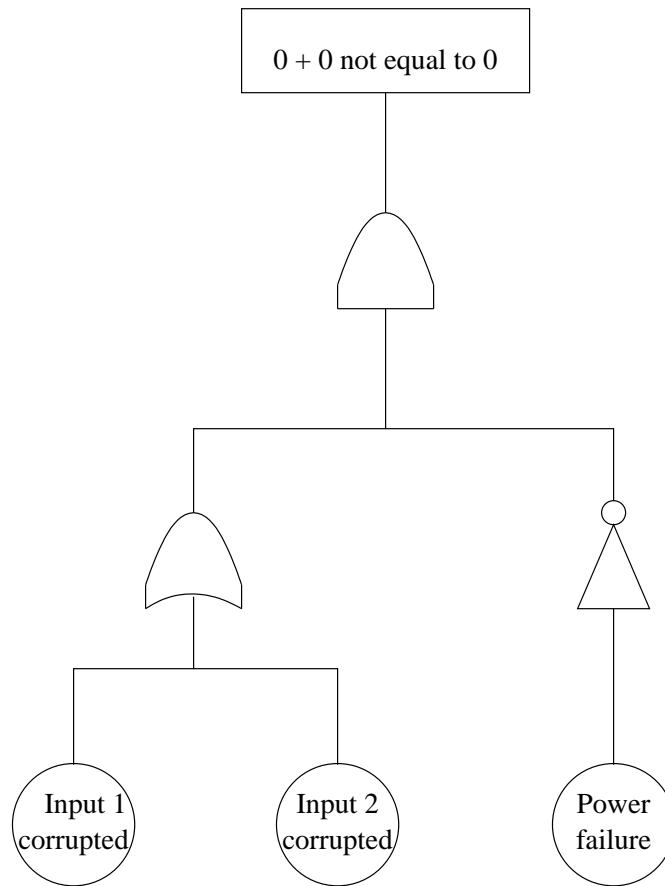
Dealing with non-monotone fault trees is harder. Indeed, it is not enough to find out which failure modes must be enabled. Some failure modes must be disabled, too. We will first present an example of such a system, then define non-monotone boolean functions, and finally describe an algorithm by Olivier Coudert and Jean-Christophe Madre [3] to perform the FTA.

4.1 Example

Below is the description of an adder:



This system is supposed to compute the sum C of two numbers A and B . Numbers are represented by finite words of bits. However, two different kinds of failures can happen: A or B get corrupted, C is forced to 0 by a power failure. The Fault Tree for the computation of $0 + 0$ follows.



Notice that it is not enough to have the inputs corrupted to get a wrong result. Indeed, a power failures may “correct” the corruption of the inputs.

4.2 Definitions

We introduce the notion of *Prime Implicants* of non-monotone functions. They play the same role as Minimal Cut Sets for monotone functions.

Assignment to boolean variables

Let V be a set of boolean variables. Then an assignment is a partial function from V to $\{\text{True}, \text{False}\}$.

Prime Implicant

An Implicant of a boolean function f is an assignment to some of the variables of f that makes f true.

To define a *Prime Implicant*, we need to define a relation \subseteq over implicants:

\forall implicants $C, D : C \subseteq D$ iff $\forall x \in \text{Dom}(C) : x \in \text{Dom}(D) \wedge C(x) = D(x)$, where $\text{Dom}(C)$ denotes the domain of C

A Prime Implicant I of f is an Implicant of f such that there is no other Implicant C such that $C \subseteq I$.

For example, $\{(A, \text{True}), (B, \text{False})\}$ is an implicant of $f : (A, B, C) \rightarrow A \vee (B \wedge C)$, but is not prime. However, $\{(A, \text{True})\}$ is.

In [3], a method using *metaproducts* to represent implicants is described. A metaproduct is a string $\{x_1, \neg x_1, \epsilon\} \times \{x_2, \neg x_2, \epsilon\} \dots \{x_n, \neg x_n, \epsilon\}$. Each symbol of the string indicates if a variable should be set to True, to False, or if its value does not matter. Thus, if $x_1 \neg x_2 x_4$ is an implicant of a function f , then $x_1 \wedge \neg x_2 \wedge x_3 \wedge x_4 \Rightarrow f$ and $x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4 \Rightarrow f$.

4.3 Solution using BDDs

Binary Decision Diagrams [6] are structures representing boolean functions in a compact way. Here is a short reminder.

4.3.1 BDDs

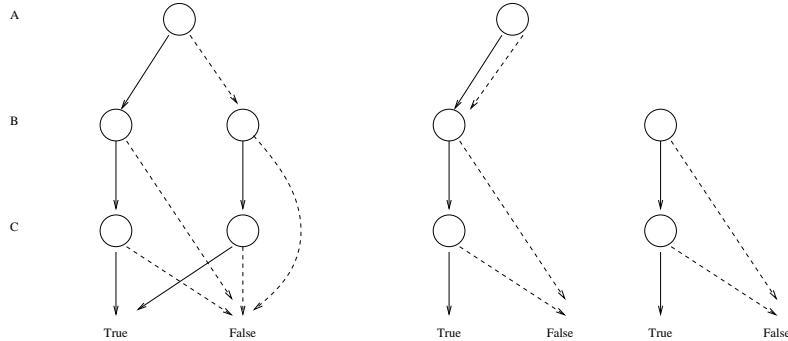
Binary Decision Diagram

A BDD is a direct acyclic graph representing a boolean expression E . Each node represents a boolean variable x_k . From each node originate either two or zero arrows. These arrows are labeled *hi* and *low*. The subgraph pointed by the *hi* arrow represents the expression E where x_k is replaced by True. Similarly, the *low* subgraph represents E where x_k is replaced by False.

Reduced Ordered Binary Decision Diagram

The order of variables encountered on all paths from the top of a BDD to the leaves must always be the same for *Ordered* Binary Decision Diagrams. A OBDD is *reduced* when the two following conditions are met:

- Two instances of a same subgraph can never exist. Instead, different nodes whose children are identical will point to the same instance of a unique child.
- There is no node such that its *hi* and *low* arrows point to the same subgraph.



The left most graph shows a simple BDD. It is not reduced, as it breaks both rules. The middle graph respects the first rule, but not the second. The right-most BDD is reduced, as it respects both rules.

From now on, the term BDD will be used, but it should really be understood as ROBDD.

4.3.2 Finding the Prime Implicants using BDDs

Here, we describe a method based on BDDs to compute and store prime implicants. For each fault variable x_i , two new variables are created in the BDD representing the set of prime implicants.

- Ox_i indicates if the variable must be set to some value, or if it can be left free
- Sx_i indicates the value x_n should take. It is only needed in the case Ox_i is true.

An efficient method to compute all the prime implicants is presented in [3] and [7]. It is based on this theorem:

$$PI(f) = \neg Ox_i \wedge PI(f_{x_i} \wedge f_{\neg x_i}) \vee (Ox_i \wedge \neg PI(f_{x_i} \wedge f_{\neg x_i})) \wedge (Sx_i \wedge PI(f_{x_i}) \vee \neg Sx_i \wedge PI(f_{\neg x_i}))$$

$PI(f)$ is the set of all the prime implicants of f .

f_x is the boolean function whose expression is the same as the one of f , but with every occurrence of x replaced by *true*.

$f_{\neg x}$ is the boolean function whose expression is the same as the one of f , but with every occurrence of x replaced by *false*.

If f is represented with a BDD, this method takes a time linear to the number of its nodes.

5 Future work

There are a number of limitations to the application of the methods in sections 3.3 and 4.3.2:

- The FTA is performed on one state of the system. Configurations variables must be assigned fixed values
- Inputs and outputs of the system must be booleans
- Sequential systems are not supported

All these limitations restrict the usefulness our the methods previously presented to a few systems. In this section, we expose what extensions should be handled, as well as the issues they raise.

5.1 Non-fixed configuration

Usually, Fault Tree Analyses are performed with fixed values assigned to the configuration variables. However, it could be interesting to be able to answer these questions:

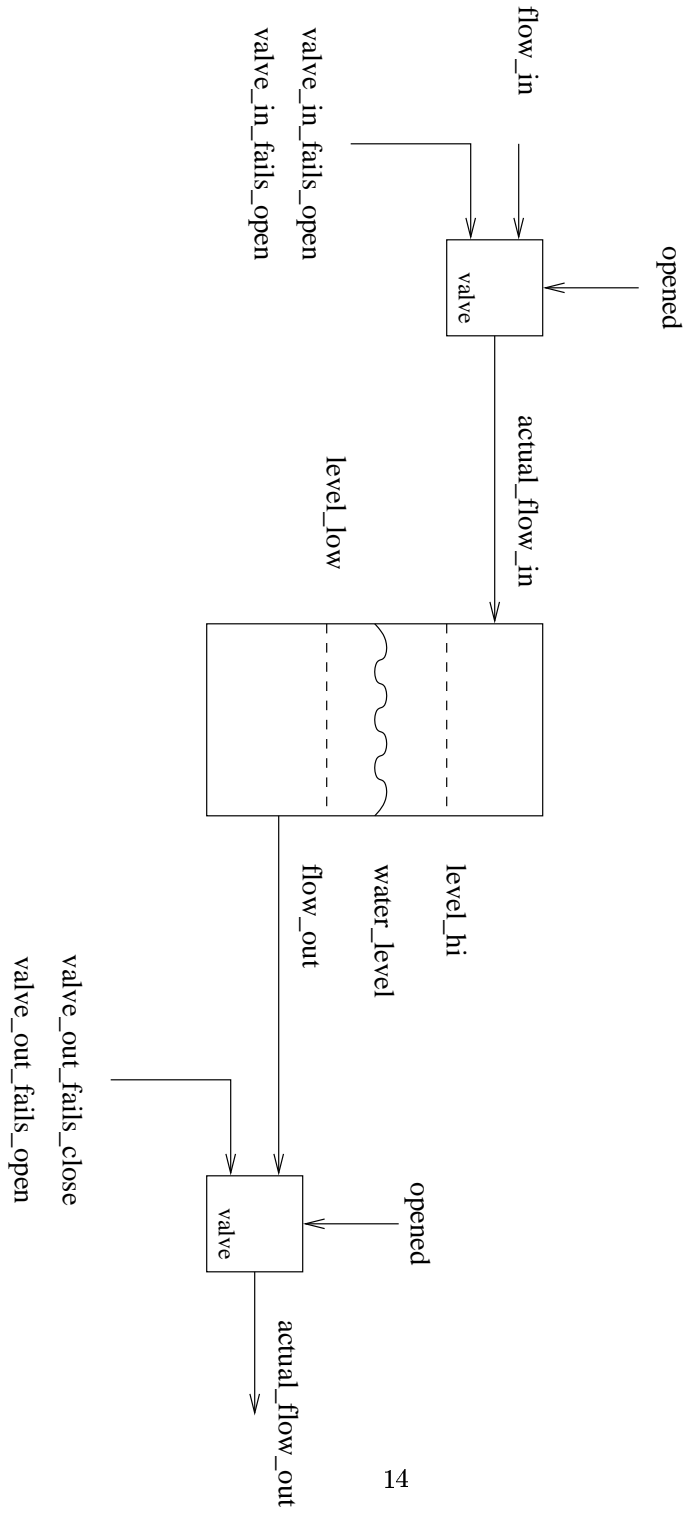
- Are there prime implicants valid in all possible configurations ?
- Generally, in which configurations a combination of failures is a prime implicant ?

How to express such prime implicants ? A solution is not to distinguish between failure and configuration variables. Recall failure modes are inputs that cannot be controlled by the user or the system. If we do not fix the value of configuration variables, they too become “uncontrollable”. New issues would then arise:

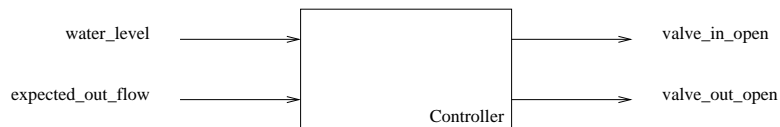
- The number of variables would be increased. This would obviously reduce the efficiency of the algorithms.
- Introducing these configuration variables into a previously monotone fault tree may break its monotonicity, thus forbidding the use of our SAT-Solver based method.
- While fault variables are always boolean, this may not be the case for inputs and outputs. For example, they could be numerical values. In that case, it is interesting to integrate arithmetics in the process. Some SAT Solvers can handle arithmetics.

Minimal Cut Sets or Prime Implicants may include predicates. For example, we encounter cut sets like { “water_level > level_hi” , valve_out_fails_close }.

Below is an example to illustrate the use of non-fixed configurations.



This system describes a tank, containing a varying level of liquid. Two valves control the amount of water to enter and to leave the tank. Those two valves can fail to open or close. For example, a valve which fails to open will remain closed. Valves cannot fail both to close and to open. Users can ask for water to be delivered. They do not have direct access to the tank. Instead, a controller handles their requests.



The controller must ensure that there always is enough water in the tank to fulfill users' requests. It must also prevent any overflow of the tank.

The goal of the analysis is to find out whether the controller will be able to control the level of water. For example, it must be able to close the input valve when no one is taking water out of the tank. It should also open the output valve in case the level of water becomes too high. While looking simple, this example combines many features:

- The level of water is defined in terms of its previous value, as its value at some point in time depends on its value a short while before. This means we are dealing with a temporal system.
- Arithmetics are used to compute the level.

One of the requirements for such a system could be “The level of water always remains between *level_lo* and *level_hi*”. Normally, a designer would have to draw several fault trees, depending on the value of *expected_out_flow* and the current level of water. Instead, it is possible to build only one fault tree. The configuration variables (*expected_out_flow* and *water_level*, in this case) play the same role as failure variables.

5.2 Sequential systems

A sequential system depends on time. Each variable can have different values, depend on the value of time. Time is discrete, and variables are only allowed to change when time changes. Such systems can be described using Lustre [10]. So far, we considered the case of combinational systems. Now we discuss how time influences FTA, and we investigate how to automate FTA of sequential systems.

We see two kind of approaches:

- Fault variables are not allowed to change. However, inputs, outputs and internal variables may change.
In the context of FTA, an implicant becomes a constant assignment to variables inducing a failure at some point in time.
- Every variable, including the fault variables, may change.
In the context of FTA, an implicant becomes a “temporal” assignment to variables inducing a failure at some point in time. This

case introduces a new difficulty. If the amount of time to pass is not bound, the number of implicants may be infinite.

We have to differentiate different kinds of failures:

- Failures that remain constant once triggered
- Temporary failures that are enabled for a limited period of time
- Temporary failures triggered regularly.

We have several possibilities concerning the configuration variables of the system:

- Provide an initial state for the system
- Put restrictions on the initial states
- Restrict how variables can change with time.

We have to deal with temporal logic. If we do not allow fault variables to vary, we could certainly use model checking to perform FTA. Both BDDs and SAT-Solvers can be used in model checking.

6 Quantitative analysis

All the work described previously deals with the qualitative analysis part of FTA. While this first analysis already provides meaningful information, designers often want to compute an estimation of the probability of occurrence of a system-level failure. In order to perform such an estimation, the designer must provide the probability of failure for each component. Assuming each component failure is independent of all other failures, the computation of the system-level failure probability can be done using a simple formula.

$$\begin{aligned} P(T) = & \sum_{i=1}^n P(C_i) \\ & - \sum_i \sum_j P(C_i \cap C_j) \\ & + \sum_i \sum_j \sum_k P(C_i \cap C_j \cap C_k) \\ & + \dots \\ & + (-1)^{n+1} P(C_1 \cap C_2 \cap \dots \cap C_n) \end{aligned}$$

Here, C_i are the prime implicants which induce the top event T . Note that this formula is made of an exponential number of terms. The need for such a number of terms comes from the fact that the prime implicants are not necessary disjoint.

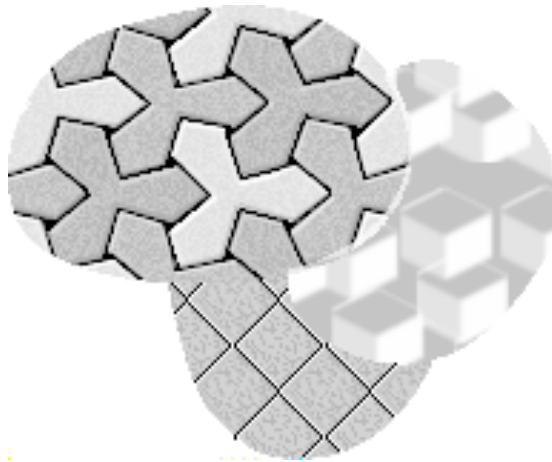


This figure shows that some areas are counted multiple times after the first step of the sum. The following terms aim at counting each area exactly once.

However, if we instead have a partition of the set of truth values of T , the formula turns into a much smaller one:

$$P(T) = \sum_{i=1}^n P(Q_i)$$

Indeed, as shown on the picture below, only the first step of the sum is now needed, as we re-arranged the three original surfaces. They are now disjoint, and each zone is counted exactly once.



In the second formula, the Q_i are the elements of the partition. The problem consists now of finding such a partition. [9] shows how BDDs can be used to manage this. Recall that a fault tree is in fact a boolean formula, which can be encoded as a BDD. Each path from the top node to the leaves in a BDD describes an implicant of this function. All those

implicants have the property that they are contradictory with every other implicant:

$$\forall i, j, i \neq j, C_i \wedge C_j = \text{false}.$$

7 Related work

We shortly introduce a few papers dealing with FTA.

[1] introduces safety analysis, FTA. It demonstrates how to use a formal methods tool to analyze non monotone systems. The tool used in this paper is NP-Tools, from Prover Technology AB.

Olivier Coudert and Jean-Christophe Madre worked on computing minimal cut sets and prime implicants of boolean functions. In [2], they present methods to compute prime covers of functions. In [3], they show an efficient method to compute the set of all prime implicants of a boolean function.

BDDs are used in both these papers. Unfortunately, some systems cannot be modeled using BDDs. In [4], Poul Frederick Williams, Macha Nikolskaïa, Antoine Rauzy show how to avoid their use in reliability analysis. Instead of BDDs, they use Boolean Expression Diagrams ([5]).

In [8], the case of systems depending on time is discussed. It is shown how to combine different methods to analyze such systems. BDDs are used to analyze systems, or parts of systems, that do not depend on time. A difference is made between static Fault Trees, and dynamic Fault Trees. Dynamic Fault Trees are used to analyze systems including a time dependency. Markov chains are used to model dynamic Fault Trees. Differential equations are derived from these Markov chains, and are solved.

Finally, J.D Andrews and S.J. Dunnett show how BDDs can be used to perform exact quantitative FTA in [9] (the section “Qualitative Analysis” in this paper is a short summary of this paper).

8 Conclusion

Our work shows that it is possible to bypass the construction of fault trees by hand when performing FTA. In order to achieve that, the system to analyze must be formally described. There are tools that aim at easing the process of formally describing a system. For example, Telelogic Tau Scade provides a graphical interface to the Lustre programming language.

We presented a method to iteratively compute minimal cut sets of monotone fault trees by using a SAT-Solver. For other non-monotone systems, it is possible to use a BDD-based method [3] by Olivier Coudert and Jean Christophe Madre.

However, these methods have their limitations. We showed extensions to traditional FTA that could be useful to designers.

References

- [1] Ove Åkerlund, Gunnar Stålmårck, Mary Helander: *Reliability Analysis for Non Monotonic System Failure Functions*.
- [2] Olivier Coudert, Jean Christophe Madre: *Implicit Prime Cover Computation: An Overview*
- [3] Olivier Coudert, Jean Christophe Madre: *Fault Tree Analysis: 10²⁰ Prime Implicants and Beyond*
- [4] Poul Frederick Williams, Macha Nikolskaïa, Antoine Rauzy: *Bypassing BDD Construction for Reliability Analysis*
- [5] H.R. Andersen and H. Hulgaard: *Boolean Expression Diagrams*
- [6] Randal E. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*
- [7] Vasco M. Manquinho, Arlindo L. Oliveira, João P. Marques Silva: *Models and Algorithms for Computing Minimum-Size Prime Implicants*
- [8] Ragavan Manian, Joanne Bechta Dugan, David Coppit, Jevin J. Sullivan: *Combining Various Solution Techniques for Dynamic Fault Tree Analysis of Computer Systems*
- [9] J.D. Andrews and S.J. Dunnett: *Event Tree Analysis Using Binary Decision Diagrams*.
- [10] N. Halbwegs, P. Caspi, P. Raymond, D. Pilaud: *The synchronous dataflow programming language LUSTRE*